Computational Logic
A “Hands-on” Introduction to (Pure) Logic Programming

Note: slides with executable links. Follow the run example links to execute the example code.
Syntax: Terms (Variables, Constants, and Structures)

(Using Prolog notation conventions)

- **Variables:** start with uppercase character (or “_”), may include “_” and digits:
  
  *Examples:* X, Im4u, A_little_garden, _, _x, _22

- **Constants:** lowercase first character, may include “_” and digits. Also, numbers and some special characters. Quoted, any character:
  
  *Examples:* a, dog, a_big_cat, 23, ’Hungry man’, []

- **Structures:** a **functor** (the structure name, is like a constant name) followed by a fixed number of arguments between parentheses:
  
  *Example:* date(monday, Month, 1994)

  Arguments can in turn be variables, constants and structures.

  ◦ **Arity:** is the number of arguments of a structure. Functors are represented as **name/arity**. A constant can be seen as a structure with arity zero.

Variables, constants, and structures as a whole are called **terms** (they are the terms of a “first–order language”): the **data structures** of a logic program.
Syntax: Terms

(Using Prolog notation conventions)

- Examples of terms:

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
<th>Main functor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dad</td>
<td>constant</td>
<td>dad/0</td>
</tr>
<tr>
<td>time(min, sec)</td>
<td>structure</td>
<td>time/2</td>
</tr>
<tr>
<td>pair(Calvin, tiger(Hobbes))</td>
<td>structure</td>
<td>pair/2</td>
</tr>
<tr>
<td>Tee(Alf, rob)</td>
<td>illegal</td>
<td>—</td>
</tr>
<tr>
<td>A_good_time</td>
<td>variable</td>
<td>—</td>
</tr>
</tbody>
</table>

- A variable is **free** if it has not been assigned a value yet.
- A term is **ground** if it contains no free variables.
- **Functors** can be defined as **prefix**, **postfix**, or **infix** operators (just syntax!):

<table>
<thead>
<tr>
<th>a + b is the term ’+’(a,b) if +/2 declared infix</th>
</tr>
</thead>
<tbody>
<tr>
<td>- b is the term ’-’(b) if -/1 declared prefix</td>
</tr>
<tr>
<td>a &lt; b is the term ’&lt;’(a,b) if &lt;/2 declared infix</td>
</tr>
</tbody>
</table>

  | john father mary is the term father(john,mary) if father/2 declared infix |

We assume that some such operator definitions are always preloaded.
Syntax: Rules and Facts (Clauses)

- **Rule:** an expression of the form:
  
  \[
  p_0(t_1, t_2, \ldots, t_{n_0}) :\leftarrow p_1(t_1^1, t_2^1, \ldots, t_{n_1}^1),
  \ldots, p_m(t_1^m, t_2^m, \ldots, t_{n_m}^m).
  \]

  ◊ \(p_0(\ldots)\) to \(p_m(\ldots)\) are *syntactically* like terms.
  ◊ \(p_0(\ldots)\) is called the **head** of the rule.
  ◊ The \(p_i\) to the right of the arrow are called **literals** and form the **body** of the rule. They are also called **procedure calls**.
  ◊ Usually, \(\leftarrow\) is called the **neck** of the rule.

- **Fact:** an expression of the form \(p(t_1, t_2, \ldots, t_n)\) (i.e., a rule with empty body).

  *Example:*
  
  meal(soup, beef, coffee). % <- A fact.
  meal(First, Second, Third) :\leftarrow
    appetizer(First),
    main_dish(Second),
    dessert(Third). % <- A rule.

  ◆ Rules and facts are both called **clauses**.
Syntax: Predicates, Programs, and Queries

- **Predicate** (or *procedure definition*): a set of clauses whose heads have the same name and arity (called the **predicate name**).

  Examples:

  ```
  pet(spot).
  pet(X) :- animal(X), barks(X).
  pet(X) :- animal(X), meows(X).
  animal(tim).
  animal(spot).
  animal(hobbes).
  ```

  Predicate `pet/1` has three clauses. Of those, one is a fact and two are rules. Predicate `animal/1` has three clauses, all facts.

- **Logic Program**: a set of predicates.

- **Query**: an expression of the form:

  `(i.e., a clause without a head).
  A query represents a question to the program.
  Example: `?- pet(X).`
“Declarative” Meaning of Facts and Rules

The declarative meaning is the corresponding one in first order logic, according to certain conventions:

- **Facts**: state things that are true.
  (Note that a fact “p.” can be seen as the rule “p :- true.”)
  
  Example: the fact `animal(spot).` can be read as “spot is an animal”.

- **Rules**:
  
  - Commas in rule bodies represent conjunction, and “:-” represents logical implication (backwards, i.e., if).
  
  - i.e., `p :- p_1, \cdots, p_m.` represents `p \leftarrow p_1 \land \cdots \land p_m.`

  Thus, a rule `p :- p_1, \cdots, p_m.` means “if `p_1` and . . . and `p_m` are true, then `p` is true”

  Example: the rule `pet(X) :- animal(X), barks(X).` can be read as “X is a pet if it is an animal and it barks”.

- Variables in facts and rules are universally quantified, \( \forall \) (recall *clausal form*).
“Declarative” Meaning of Predicates and Queries

- **Predicates**: clauses in the same predicate
  
  \[
  p : - p_1, \ldots, p_n \\
  p : - q_1, \ldots, q_m \\
  \ldots
  \]

  provide different *alternatives* (for \( p \)).

*Example*: the rules

```prolog
pet(X) : - animal(X), barks(X).
pet(X) : - animal(X), meows(X).
```

express two *alternative* ways for \( X \) to be a pet.

- **Note** (variable *scope*): the \( X \) vars. in the two clauses above are different, despite the same name. Vars. are *local to clauses* (and are *renamed* any time a clause is used—as with vars. local to a procedure in conventional languages).

- **A query** represents a *question to the program*.

*Examples:*

```prolog
?- pet(spot).
?- pet(X).
```

Asks: Is spot a pet?  Asks: “Is there an \( X \) which is a pet?”
“Execution” and Semantics

• Example of a logic program:

```
pet(X) :- animal(X), barks(X).
pet(X) :- animal(X), meows(X).
animal(tim).    barks(spot).
animal(spot).   meows(tim).
animal(hobbes). roars(hobbes).
```

• Execution: given a program and a query, executing the logic program is attempting to find an answer to the query.

Example: given the program above and the query `?- pet(X).`
the system will try to find a “substitution” for X which makes pet(X) true.

◊ The declarative semantics specifies what should be computed (all possible answers).
  ⇒ Intuitively, we have two possible answers: \( X = \text{spot} \) and \( X = \text{tim} \).

◊ The operational semantics specifies how answers are computed (which allows us to determine how many steps it will take).
Running Programs in a Logic Programming System

- Interaction with the system query evaluator (the “top level”):

```prolog
Ciao X.Y-...
?- use_module(pets).
yes
?- pet(spot).
yes
?- pet(X).
X = spot ? ;
X = tim ? ;
no
?- 
```

See the part on Developing Programs with a Logic Programming System for more details on the particular system used in the course (Ciao).
Simple (Top-Down) Operational Meaning of Programs

- A logic program is operationally a set of *procedure definitions* (the predicates).
- A query `?- p` is an initial *procedure call*.
- A procedure definition with one *clause* `p :- p_1, ..., p_m`. means:
  “to execute a call to `p` you have to *call* `p_1` and ... and `p_m`”
  ◦ In principle, the order in which `p_1, ..., p_n` are called does not matter, but, in practical systems it is fixed.
- If several clauses (definitions) `p :- p_1, ..., p_n` means:
  `p :- q_1, ..., q_m`
  “to execute a call to `p`, call `p_1` and ... and `p_n`, or, alternatively, `q_1` and ... and `q_m`, or ...”
  ◦ Unique to logic programming –it is like having several alternative procedure definitions.
  ◦ Means that several possible paths may exist to a solution and they *should be explored*.
  ◦ System usually stops when the first solution found, user can ask for more.
  ◦ Again, in principle, the order in which these paths are explored does not matter (*if certain conditions are met*), but, for a given system, this is typically also fixed.

In the following we define a more precise operational semantics.
Unification: uses

- **Unification** is the mechanism used in *procedure calls* to:
  - Pass parameters.
  - “Return” values.
- It is also used to:
  - Access parts of structures.
  - Give values to variables.
- **Unification** is a procedure to **solve equations on data structures**.
  - As usual, it returns a minimal solution to the equation (or the equation system).
  - As many equation solving procedures it is based on isolating variables and then *instantiating* them with their values.
Unification

- **Unifying two terms (or literals) $A$ and $B$**: is asking if they can be made syntactically identical by giving (minimal) values to their variables.
  - I.e., find a **variable substitution** $\theta$ such that $A\theta = B\theta$ (or, if impossible, *fail*).
  - Only variables can be given values!
  - Two structures can be made identical only by making their arguments identical.

**E.g.**:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\theta$</th>
<th>$A\theta$</th>
<th>$B\theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>dog</td>
<td>dog</td>
<td>$\emptyset$</td>
<td>dog</td>
<td>dog</td>
</tr>
<tr>
<td>$X$</td>
<td>$a$</td>
<td>${X=a}$</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>$X$</td>
<td>$Y$</td>
<td>${X=Y}$</td>
<td>$Y$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$f(X, g(t))$</td>
<td>$f(m(h), g(M))$</td>
<td>${X=m(h), M=t}$</td>
<td>$f(m(h), g(t))$</td>
<td>$f(m(h), g(t))$</td>
</tr>
<tr>
<td>$f(X, g(t))$</td>
<td>$f(m(h), t(M))$</td>
<td>Impossible (1)</td>
<td>$f(m(h), g(t))$</td>
<td>$f(m(h), g(t))$</td>
</tr>
<tr>
<td>$f(X, X)$</td>
<td>$f(Y, l(Y))$</td>
<td>Impossible (2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- (1) Structures with different name and/or arity cannot be unified.
- (2) A variable cannot be given as value a term which contains that variable, because it would create an infinite term. This is known as the **occurs check**.
  (See, however, **cyclic terms** later.)
Unification

- Often several solutions exist, e.g.:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\theta_1$</th>
<th>$A\theta_1$ and $B\theta_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(X, g(T))$</td>
<td>$f(m(H), g(M))$</td>
<td>${ X=m(a), H=a, M=b, T=b }$</td>
<td>$f(m(a), g(b))$</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
<td>${ X=m(H), M=f(A), T=f(A) }$</td>
<td>$f(m(H), g(f(A)))$</td>
</tr>
</tbody>
</table>

These are correct, but a simpler ("more general") solution exists:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\theta_1$</th>
<th>$A\theta_1$ and $B\theta_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(X, g(T))$</td>
<td>$f(m(H), g(M))$</td>
<td>${ X=m(H), T=M }$</td>
<td>$f(m(H), g(M))$</td>
</tr>
</tbody>
</table>

- Always a unique (modulo variable renaming) most general solution exists (unless unification fails).
- This is the one that we are interested in.
- The unification algorithm finds this solution.
Unification Algorithm

- Select one equation from the equation system, delete it, and, depending on the form of the equation:
  
  - $X=X$ : ignore
  
  - $X=f(\ldots, X, \ldots)$ : fail (occurs check)
  
  - $X=\text{term}$ :
    
    * add it to the solution
    
    * replace $X$ by $\text{term}$ anywhere else
  
  - $a=a$ : ignore
  
  - $a=b$ : fail
  
  - $a=f(\ldots)$ : fail
  
  - $g(\ldots)=f(\ldots)$ : fail
  
  - $f(\ldots m \ldots)=f(\ldots n \ldots)$ ($m \neq n$) : fail
  
  - $f(s_1, \ldots, s_n)=f(t_1, \ldots, t_n)$ :
    
    * add to the system: $s_1=t_1, \ldots, s_n=t_n$
Unification Algorithm Examples

- Unify: \( p(X, f(b)) \) and \( p(a, Y) \)
  \[
  \begin{align*}
  p(X, f(b)) &= p(a, Y) \\
  X &= a \\
  Y &= f(b)
  \end{align*}
  \]

- Unify: \( p(X, f(Y)) \) and \( p(a, g(b)) \)
  \[
  \begin{align*}
  p(X, f(Y)) &= p(a, g(b)) \\
  X &= a \\
  f(Y) &= g(b) \\
  \end{align*}
  \]
  fail

- Unify: \( p(X, X) \) and \( p(f(Z), f(W)) \)
  \[
  \begin{align*}
  p(X, X) &= p(f(Z), f(W)) \\
  X &= f(Z) \\
  X &= f(Z) \\
  X &= f(Z) \\
  Z &= W \\
  Z &= W
  \end{align*}
  \]

- Unify: \( p(X, f(Y)) \) and \( p(Z, X) \)
  \[
  \begin{align*}
  p(X, f(Y)) &= p(Z, X) \\
  X &= Z \\
  f(Y) &= X \\
  f(Y) &= Z \\
  f(Y) &= X \\
  f(Y) &= Z \\
  \end{align*}
  \]

- Unify: \( p(X, f(X)) \) and \( p(Z, Z) \)
  \[
  \begin{align*}
  p(X, f(X)) &= p(Z, Z) \\
  X &= Z \\
  X &= Z \\
  f(X) &= Z \\
  f(Z) &= Z \\
  \end{align*}
  \]
  fail (“Occurs check”)
Unification Algorithm 2 (Just A More Formal Version)

- Let $A$ and $B$ be two terms:

1. $\theta = \emptyset$, $E = \{A = B\}$

2. while not $E = \emptyset$:

   2.1 delete an equation $T = S$ from $E$

   2.2 case $T$ or $S$ (or both) are (distinct) variables. Assuming $T$ variable:

      * (occur check) if $T$ occurs in the term $S$ → halt with failure
      * substitute variable $T$ by term $S$ in all terms in $\theta$
      * substitute variable $T$ by term $S$ in all terms in $E$
      * add $T = S$ to $\theta$

   2.3 case $T$ and $S$ are non-variable terms:

      * if their names or arities are different → halt with failure
      * obtain the arguments $\{T_1, \ldots, T_n\}$ of $T$ and $\{S_1, \ldots, S_n\}$ of $S$
      * add $\{T_1 = S_1, \ldots, T_n = S_n\}$ to $E$

3. halt with $\theta$ being the m.g.u of $A$ and $B$
Unification Algorithm 2 - Examples (I)

- Unify: \( A = p(X, X) \) and \( B = p(f(Z), f(W)) \)

\[
\begin{array}{cccc}
\theta & E & T & S \\
\emptyset & \{ p(X, X) = p(f(Z), f(W)) \} & p(X, X) & p(f(Z), f(W)) \\
\emptyset & \{ X = f(Z), X = f(W) \} & X & f(Z) \\
\{ X = f(Z) \} & \{ f(Z) = f(W) \} & f(Z) & f(W) \\
\{ X = f(Z) \} & \{ Z = W \} & Z & W \\
\{ X = f(W), Z = W \} & & & \\
\end{array}
\]

- Unify: \( A = p(X, f(Y)) \) and \( B = p(Z, X) \)

\[
\begin{array}{cccc}
\theta & E & T & S \\
\emptyset & \{ p(X, f(Y)) = p(Z, X) \} & p(X, f(Y)) & p(Z, X) \\
\emptyset & \{ X = Z, f(Y) = X \} & X & Z \\
\{ X = Z \} & \{ f(Y) = Z \} & f(Y) & Z \\
\{ X = f(Y), Z = f(Y) \} & & & \\
\end{array}
\]
Unification Algorithm 2 - Examples (II)

- Unify: \( A = p(X, f(Y)) \) and \( B = p(a, g(b)) \)

<table>
<thead>
<tr>
<th>( \theta )</th>
<th>( E )</th>
<th>( T )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ p(X, f(Y)) = p(a, g(b)) }</td>
<td>p(X, f(Y))</td>
<td>p(a, g(b))</td>
</tr>
<tr>
<td>{ X = a, f(Y) = g(b) }</td>
<td>X</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>{ X = a }</td>
<td>{ f(Y) = g(b) }</td>
<td>f(Y)</td>
<td>g(b)</td>
</tr>
</tbody>
</table>

fail

- Unify: \( A = p(X, f(X)) \) and \( B = p(Z, Z) \)

<table>
<thead>
<tr>
<th>( \theta )</th>
<th>( E )</th>
<th>( T )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ p(X, f(X)) = p(Z, Z) }</td>
<td>p(X, f(X))</td>
<td>p(Z, Z)</td>
</tr>
<tr>
<td>{ X = Z, f(X) = Z }</td>
<td>X</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>{ X = Z }</td>
<td>{ f(Z) = Z }</td>
<td>f(Z)</td>
<td>Z</td>
</tr>
</tbody>
</table>

fail
A (Schematic) Interpreter for Logic Programs (SLD–resolution)

**Input:** A logic program $P$, a query $Q$

**Output:** $\mu$ (answer substitution) if $Q$ is provable from $P$, *failure* otherwise

1. Make a copy $Q'$ of $Q$
2. Initialize the “resolvent” $R$ to be $\{Q\}$
3. While $R$ is nonempty do:
   3.1. Take a literal $A$ in $R$
   3.2. Take a clause $A':-B_1,\ldots,B_n$ (*renamed*) from $P$ with $A'$ same predicate symbol as $A$
      3.2.1. If there is a solution $\theta$ to $A = A'$ (*unification*)
          • Replace $A$ in $R$ by $B_1,\ldots,B_n$
          • Apply $\theta$ to $R$ and $Q$
      3.2.2. Otherwise, take another clause and repeat
3.3. If there are no more clauses, go back to some other choice
3.4. If there are no pending choices left, output *failure*

4. ($R$ empty) Output solution $\mu$ to $Q = Q'$
5. Explore another pending branch for more solutions (upon request)
A (Schematic) Interpreter for Logic Programs (Standard Prolog)

**Input:** A logic program $P$, a query $Q$

**Output:** $\mu$ (answer substitution) if $Q$ is provable from $P$, failure otherwise

1. Make a copy $Q'$ of $Q$
2. Initialize the “resolvent” $R$ to be \{ $Q$ \}
3. While $R$ is nonempty do:
   3.1. Take the **leftmost** literal $A$ in $R$
   3.2. Take the **first** clause $A':-B_1, \ldots, B_n$ (renamed) from $P$
       with $A'$ same predicate symbol as $A$
       3.2.1. If there is a solution $\theta$ to $A = A'$ (**unification**)
           - Replace $A$ in $R$ by $B_1, \ldots, B_n$
           - Apply $\theta$ to $R$ and $Q$
       3.2.2. Otherwise, take the **next** clause and repeat
   3.3. If there are no more clauses, go back to *most recent pending choice*
   3.4. If there are no pending choices left, output failure
4. ($R$ empty) Output solution $\mu$ to $Q = Q'$
5. Explore the **most recent** pending branch for more solutions (upon request)
• Step 3.2 defines *alternative paths* to be explored to find answer(s); execution explores this tree (for example, breadth-first).

• Since step 3.2 is left open, a given logic *programming* system must specify how it deals with this by providing one (or more)

  ◦ **Search rule(s):** “how are clauses/branches selected in 3.2.”

• Note that choosing a different clause (in step 3.2) can lead to finding solutions in a different order – Example (two valid executions):

  ```
  ?- pet(X).
  X = spot ? ;
  X = tim ? ;
  no
  ?- ?-
  ?- pet(X).
  X = tim ? ;
  X = spot ? ;
  no
  ?- ?-
  ```

• In fact, there is also some freedom in step 3.1, i.e., a system may also specify:

  ◦ **Computation rule(s):** “how are literals selected in 3.1.”
Running Programs: Alternative Execution Paths

C₁: \( \text{pet}(X) \) :- \( \text{animal}(X) \), \( \text{barks}(X) \).
C₂: \( \text{pet}(X) \) :- \( \text{animal}(X) \), \( \text{meows}(X) \).
C₃: \( \text{animal}(\text{tim}) \).
C₄: \( \text{animal}(\text{spot}) \).
C₅: \( \text{animal}(\text{hobbes}) \).
C₆: \( \text{barks}(\text{spot}) \).
C₇: \( \text{meows}(\text{tim}) \).
C₈: \( \text{roars}(\text{hobbes}) \).

- \( \text{?- pet}(X) \). \ (\text{top-down, left-to-right})

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( R )</th>
<th>Clause</th>
<th>( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{pet}(X) )</td>
<td>( \text{pet}(X) )</td>
<td>( C₁ ) *</td>
<td>{ X=X₁ }</td>
</tr>
<tr>
<td>( \text{pet}(X₁) )</td>
<td>( \text{animal}(X₁), \text{barks}(X₁) )</td>
<td>( C₃ ) *</td>
<td>{ X₁=\text{tim} }</td>
</tr>
<tr>
<td>( \text{pet}(\text{tim}) )</td>
<td>( \text{barks}(\text{tim}) )</td>
<td>???</td>
<td>failure</td>
</tr>
</tbody>
</table>

* means *choice-point*, i.e., other clauses applicable.

- But solutions exist in other paths!
Running Programs: Alternative Execution Paths

\[ C_1: \text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X). \]
\[ C_2: \text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X). \]
\[ C_3: \text{animal}(\text{tim}). \quad C_6: \text{barks}(\text{spot}). \]
\[ C_4: \text{animal}(\text{spot}). \quad C_7: \text{meows}(\text{tim}). \]
\[ C_5: \text{animal}(\text{hobbes}). \quad C_8: \text{roars}(\text{hobbes}). \]

\[ \text{?- pet}(X). \] (top-down, left-to-right)

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( R )</th>
<th>Clause</th>
<th>( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(X)</td>
<td>pet(X)</td>
<td>( C_1^* )</td>
<td>( { X=X_1 } )</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), barks(X_1)</td>
<td>( C_3^* )</td>
<td>( { X_1=\text{tim} } )</td>
</tr>
<tr>
<td>pet(tim)</td>
<td>barks(tim)</td>
<td>???</td>
<td>failure</td>
</tr>
</tbody>
</table>

* means choice-point, i.e., other clauses applicable.

- But solutions exist in other paths!

→ Let’s go back to our last choice point (\( C_3^* \)) and try the next alternative...
Running Programs: Alternative Execution Paths

C1: pet(X) :- animal(X), barks(X).
C2: pet(X) :- animal(X), meows(X).
C3: animal(tim).
C4: animal(spot).
C5: animal(hobbes).
C6: barks(spot).
C7: meows(tim).
C8: roars(hobbes).

• ?- pet(X). (top-down, left-to-right, different branch)

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( R )</th>
<th>Clause</th>
<th>( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(X)</td>
<td>pet(X)</td>
<td>C1*</td>
<td>{ X=X_1 }</td>
</tr>
<tr>
<td>pet(X_1)</td>
<td>animal(X_1), barks(X_1)</td>
<td>C4*</td>
<td>{ X_1=spot }</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>barks(spot)</td>
<td>C6</td>
<td>{}</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

• System response: \( X=\text{spot} \) ?
• If we type “;” after the ? prompt (i.e., we ask for another solution) the system can go and execute a different branch (i.e., a different choice in C4*, or C1*).
The Search Tree Revisited

- Different execution strategies explore the tree in a different way.
- A strategy is complete if it guarantees that it will find all existing solutions.
The Search Tree Revisited

- Different execution strategies explore the tree in a different way.
- A strategy is complete if it guarantees that it will find all existing solutions.
- Standard Prolog does it top-down, left-to-right (i.e., depth-first).

```
pet(X) :- animal(X), barks(X).
pet(X) :- animal(X), meows(X).
```

```
pet(X) :- animal(tim), barks(spot).
pet(X) :- animal(spot).
animal(hobbes).
barks(spot).
```

```
animal(X), barks(X) animal(X), meows(X)
pet(X)
X=tim X=spot X=hobbes X=tim X=spot X=hobbes
failfail fail failsolution solution
```
• All solutions are at finite depth in the tree.
• Failures can be at finite depth or, in some cases, be an infinite branch.
Depth-First Search (Backtracking)

- Incomplete: may fall through an infinite branch before finding all solutions.
- But very efficient: it can be implemented with a call stack, very similar to a traditional programming language.
Breadth-First Search

- Will find all solutions before falling through an infinite branch.
- But costly in terms of time and memory.
- Used in all the following examples (via Ciao’s bf package).
Selecting breadth-first or depth-first search

- In the Ciao system we can select the search rule using the packages mechanism.

- Files should start with the following line:
  - To execute in *breadth-first* mode:
    ```prolog
    :- module(_,_,[sr/bfall]).
    ```
  - To execute in *depth-first* mode:
    ```prolog
    :- module(_,_,[]).
    ```

See the part on Developing Programs with a Logic Programming System for more details on the particular system used in the course (Ciao).
Control of Search in Depth-First Search (Backtracking)

Conventional programs (no search) execute conventionally.

Programs **with search**: programmer has at least three ways of *controlling search*:

1. **The ordering of literals** in the body of a clause:
   - Profound effect on the *size of the computation* (at the limit, on termination).

   Compare executing `p(X), q(X,Y)` with executing `q(X,Y), p(X)` in:

   ```
   p(X) :- X = 4.
   q(X, Y) :- X = 1, Y = a, ...
   p(X) :- X = 5.
   q(X, Y) :- X = 2, Y = b, ...
   q(X, Y) :- X = 4, Y = c, ...
   q(X, Y) :- X = 4, Y = d, ...
   ```

   `p(X), q(X,Y)` is more efficient: execution of `p/2` reduces the choices of `q/2`.

   - Note that optimal order depends on the variable instantiation mode:
     E.g., for `q(X,d), p(X)`, this order is better than `p(X), q(X,d)`.
The ordering of clauses in a predicate:

- Affects the order in which solutions are generated.

  E.g., in the previous example we get: X\(=4\), Y\(=c\) as the first solution and X\(=4\), Y\(=d\) as the second.

  If we reorder q/2:

  \[
  \begin{align*}
  p(X) & :- \ X = 4. \\
p(X) & :- \ X = 5. \\
q(X, Y) & :- \ X = 4, Y = d, \ldots \\
q(X, Y) & :- \ X = 4, Y = c. \ldots \\
q(X, Y) & :- \ X = 2, Y = b, \ldots \\
q(X, Y) & :- \ X = 1, Y = a, \ldots
  \end{align*}
  \]

  we get X\(=4\), Y\(=d\) first and then X\(=4\), Y\(=c\).

- If can also affect the size of the computation and termination.

The pruning operators (e.g., “cut”), which cut choices dynamically –see later.
Role of Unification in Execution

• As mentioned before, unification used to access data and give values to variables. 

Example: Consider query ?- animal(A), named(A,Name). with: animal(dog(tim)). named(dog(Name),Name).

• Also, unification is used to pass parameters in procedure calls and to return values upon procedure exit.

<table>
<thead>
<tr>
<th>Q</th>
<th>R</th>
<th>Clause</th>
<th>θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(P)</td>
<td>pet(P)</td>
<td>C₁*</td>
<td>{ P=X₁ }</td>
</tr>
<tr>
<td>pet(X₁)</td>
<td>animal(X₁), barks(X₁)</td>
<td>C₃*</td>
<td>{ X₁=spot }</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>barks(spot)</td>
<td>C₆</td>
<td>{}</td>
</tr>
<tr>
<td>pet(spot)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
“Modes”

- In fact, argument positions are not fixed a priori to be input or output.

  **Example**: Consider query `?- pet(spot).` vs. `?- pet(X).`

  or in the Peano arithmetic example from the introduction:

  ```prolog
  ?- plus( s(0), s(s(0)), Z).  % Adds
  vs.  ?- plus( s(0), Y, s(s(s(0)))) .  % Subtracts
  ```

- Thus, procedures can be used in different **modes** s.t. different sets of arguments are input or output in each mode.

- We sometimes use `+` and `-` to refer to, respectively, and argument being an input or an output, e.g.:

  ```prolog
  plus(+X, +Y, -Z)  means we call `plus` with
  ◦ X instantiated,
  ◦ Y instantiated, and
  ◦ Z free.
  ```
Computational Logic
Pure Logic Programming Examples
Pure Logic Programs (Overview)

- Programs that only make use of unification (i.e., what we have described so far).

- They are fully “logical:”
  the set of computed answers “coincides” with the set of logical consequences.

  ◦ *Computed answers*: the answers for all queries that terminate successfully.

- Allow programming declaratively:
  describe the problem, make queries, obtain correct answers
  → specifications as programs

- They have full computational power (Turing completeness).

(Recall the initial slides for the course.)
Database Programming

- A Logic Database is a set of facts and rules (i.e., a logic program):

  
  father_of(john, peter).
  father_of(john, mary).
  father_of(peter, michael).
  mother_of(mary, david).

  grandfather_of(L, M) :- father_of(L, N), father_of(N, M).
  grandfather_of(X, Y) :- father_of(X, Z), mother_of(Z, Y).

- Given such logic database, a logic programming system can answer questions (queries) such as:

  ?- father_of(john, peter).
  yes

  ?- father_of(john, david).
  no

  ?- father_of(john, X).
  X = peter ;
  X = mary

  ?- grandfather_of(X, michael).
  X = john

  ?- grandfather_of(X, Y).
  X = john, Y = michael ;
  X = john, Y = david

  ?- grandfather_of(X, X).
  no

- Try to write the rules for grandmother_of(X, Y).

- Also for parent/2, ancestor/2, related/2 (have a common ancestor).
Another example:

resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).

\[
\begin{align*}
\text{inverter}(\text{Input}, \text{Output}) & : - \\
& \text{transistor}(\text{Input}, \text{ground}, \text{Output}), \text{resistor}(\text{power}, \text{Output}). \\
\text{nand_gate}(\text{Input1}, \text{Input2}, \text{Output}) & : - \\
& \text{transistor}(\text{Input1}, \text{X}, \text{Output}), \text{transistor}(\text{Input2}, \text{ground}, \text{X}), \text{resistor}(\text{power}, \text{Output}). \\
\text{and_gate}(\text{Input1}, \text{Input2}, \text{Output}) & : - \\
& \text{nand_gate}(\text{Input1}, \text{Input2}, \text{X}), \text{inverter}(\text{X}, \text{Output}).
\end{align*}
\]

Query \(\text{?- and_gate(In1,In2,Out)}\) has solution: \(\text{In1}=n3, \text{In2}=n5, \text{Out}=n1\)
Structured Data and Data Abstraction (and the ’=’ Predicate)

- **Data structures** are created using (complex) terms.

- Structuring data is important:
  
  course(complog,wed,18,30,20,30,'M. ','Hermenegildo',new,5102).

- When is the Computational Logic course?


- Structured version:

  course(complog,Time,Lecturer, Location) :-
  Time = t(wed,18:30,20:30),
  Lecturer = lect('M. ','Hermenegildo'),
  Location = loc(new,5102).

**Note:** “X=Y” is equivalent to “’=’(X,Y)”
where the predicate =/2 is defined as the fact “’=’(X,X).” – Plain unification!

- Equivalent to:

  course(complog, t(wed,18:30,20:30),
          lect('M. ','Hermenegildo'), loc(new,5102)).
Structured Data and Data Abstraction (and The Anonymous Variable)

- Given:

```prolog
course(complog,Time,Lecturer, Location) :-
    Time = t(wed,18:30,20:30),
    Lecturer = lect('M.' , 'Hermenegildo'),
    Location = loc(new,5102).
```

- When is the Computational Logic course?

```prolog
?- course(complog, Time, A, B).
has solution:
    Time=t(wed,18:30,20:30), A=lect('M.' , 'Hermenegildo'), B=loc(new,5102)
```

- Using the *anonymous variable* (“_”):

```prolog
?- course(complog,Time, _, _).
has solution:
    Time=t(wed,18:30,20:30)
```
Terms as Data Structures with Pointers

• **main** below is a procedure, that:
  ◦ creates some data structures, with *pointers* and *aliasing*.
  ◦ *calls* other procedures, *passing* to them *pointers* to these structures.

```prolog
main :-
  X=f(K,g(K)),
  Y=a,
  Z=g(L),
  W=h(b,L),
% Heap memory at this point->
  p(X,Y),
  q(Y,Z),
  r(W).
```

• Terms are data structures with pointers.

• Logical variables are *declarative* pointers.
  ◦ Declarative: they can only be assigned once.
Structured Data and Data Abstraction (Contd.)

- The circuit example revisited:

  ```prolog
  resistor(r1,power,n1). transistor(t1,n2,ground,n1).
  resistor(r2,power,n2). transistor(t2,n3,n4,n2).
  transistor(t3,n5,ground,n4).

  inverter(inv(T,R),Input,Output) :-
      transistor(T,Input,ground,Output),
      resistor(R,power,Output).

  nand_gate(nand(T1,T2,R),Input1,Input2,Output) :-
      transistor(T1,Input1,X,Output),
      transistor(T2,Input2,ground,X),
      resistor(R,power,Output).

  and_gate(and(N,I),Input1,Input2,Output) :-
      nand_gate(N,Input1,Input2,X), inverter(I,X,Output).
  ```

- The query

  ```prolog
  ?- and_gate(G,In1,In2,Out).
  ```

  has solution:

  ```prolog
  G=and(nand(t2,t3,r2),inv(t1,r1)),In1=n3,In2=n5,Out=n1
  ```
Logic Programs and the Relational DB Model

<table>
<thead>
<tr>
<th>Relational Database</th>
<th>Logic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation Name</td>
<td>Predicate symbol</td>
</tr>
<tr>
<td>Relation</td>
<td>Procedure consisting of ground facts (facts without variables)</td>
</tr>
<tr>
<td>Tuple</td>
<td>Ground fact</td>
</tr>
<tr>
<td>Attribute</td>
<td>Argument of predicate</td>
</tr>
</tbody>
</table>

**“Person”**

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>20</td>
<td>M</td>
</tr>
<tr>
<td>Jones</td>
<td>21</td>
<td>F</td>
</tr>
<tr>
<td>Smith</td>
<td>36</td>
<td>M</td>
</tr>
</tbody>
</table>

**“Lived in”**

<table>
<thead>
<tr>
<th>Name</th>
<th>Town</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>London</td>
<td>15</td>
</tr>
<tr>
<td>Brown</td>
<td>York</td>
<td>5</td>
</tr>
<tr>
<td>Jones</td>
<td>Paris</td>
<td>21</td>
</tr>
<tr>
<td>Smith</td>
<td>Brussels</td>
<td>15</td>
</tr>
<tr>
<td>Smith</td>
<td>Santander</td>
<td>5</td>
</tr>
</tbody>
</table>

The argnames package can be used to give names to arguments:

```prolog
:- use_package(argnames).
:- argnames person(name, age, sex).
:- argnames lived_in(name, town, years).
```

run example
The operations of the relational model are easily implemented as rules.

- **Union:** \[ r\_union\_s(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n). \]
  \[ r\_union\_s(X_1, \ldots, X_n) \leftarrow s(X_1, \ldots, X_n). \]

- **Cartesian Product:**
  \[ r \times s(X_1, \ldots, X_m, X_{m+1}, \ldots, X_{m+n}) \leftarrow r(X_1, \ldots, X_m), s(X_{m+1}, \ldots, X_{m+n}). \]

- **Selection:** \[ r\_selected(X_1, X_2, X_3) \leftarrow r(X_1, X_2, X_3), \leq(X_2, X_3). \]
  \( \leq/2 \) can be, e.g., Peano, Prolog built-in, constraints...

- **Set Difference:**
  \[ r\_diff\_s(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n), \text{not } s(X_1, \ldots, X_n). \]
  \[ r\_diff\_s(X_1, \ldots, X_n) \leftarrow s(X_1, \ldots, X_n), \text{not } r(X_1, \ldots, X_n). \]
  (we postpone the discussion on negation until later.)

Derived operations – some can be expressed more directly in LP:

- **Intersection:** \[ r\_meet\_s(X_1, \ldots, X_n) \leftarrow r(X_1, \ldots, X_n), s(X_1, \ldots, X_n). \]

- **Join:** \[ r\_join\_X2\_s(X_1, \ldots, X_n) \leftarrow r(X_1, X_2, X_3, \ldots, X_n), s(X_1', X_2, X_3', \ldots, X_n'). \]

Duplicates an issue: see “setof” later in Prolog.
Deductive Databases

- The subject of “deductive databases” uses these ideas to develop logic-based databases.
  - Often syntactic restrictions (a subset of definite programs) used (e.g. “Datalog” – no functors, no existential variables).
  - Variations of a “bottom-up” execution strategy used: Use the $T_p$ operator (explained in the theory part) to compute the model, restrict to the query.
  - Powerful notions of negation supported: S-models
    → Answer Set Programming (ASP)
    → powerful knowledge representation and reasoning systems.
Recursive Programming

- **Example:** ancestors.

  ```prolog
  parent(X,Y) :- father(X,Y).
  parent(X,Y) :- mother(X,Y).

  ancestor(X,Y) :- parent(X,Y).
  ancestor(X,Y) :- parent(X,Z), parent(Z,Y).
  ancestor(X,Y) :- parent(X,Z), parent(Z,W), parent(W,Y).
  ancestor(X,Y) :- parent(X,Z), parent(Z,W), parent(W,K), parent(K,Y).
  ...
  ```

- **Defining ancestor recursively:**

  ```prolog
  parent(X,Y) :- father(X,Y).
  parent(X,Y) :- mother(X,Y).

  ancestor(X,Y) :- parent(X,Y).
  ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

  ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
  ```

- **Exercise:** define “related”, “cousin”, “same generation”, etc.
Types

- **Type**: a (possibly infinite) set of terms.
- **Type definition**: A program defining a type.
- **Example**: Weekday:
  - Set of terms to represent: "Monday", "Tuesday", "Wednesday", ...
  - Type definition:
    ```prolog
    weekday('Monday').
    weekday('Tuesday').  ...
    ```

- **Example**: Date (weekday * day in the month):
  - Set of terms to represent: `date('Monday',23)`, `date('Tuesday',24)`, ...
  - Type definition:
    ```prolog
    date(date(W,D)) :- weekday(W), day_of_month(D).
    day_of_month(1).
    day_of_month(2).
    ...
    day_of_month(31).
    ```
Recursive Programming: Recursive Types

- **Recursive types**: defined by recursive logic programs.
- **Example**: natural numbers (simplest recursive data type):
  - Set of terms to represent: \( 0, s(0), s(s(0)), \ldots \)
  - Type definition:
    
    
    
    
    A *minimal recursive predicate*:
    one unit clause and one recursive clause (with a single body literal).

- Types are *runnable* and can be used to check or produce values:
  - \( \texttt{?- nat(X)} \Rightarrow X=0; X=s(0); X=s(s(0)); \ldots \)

- We can reason about *complexity*, for a given *class of queries* ("mode"). E.g., for mode \( \texttt{nat(ground)} \) complexity is *linear* in size of number.

- **Example**: integers:
  - Set of terms to represent: \( 0, s(0), -s(0), \ldots \)
  - Type definition:
    
    
    

Recursive Programming: Arithmetic

- Defining the natural order ($\leq$) of natural numbers:

  ```prolog
  less_or_equal(0, X) :- nat(X).
  less_or_equal(s(X), s(Y)) :- less_or_equal(X, Y).
  ```

  ◦ Multiple uses (modes):
    ```prolog
    less_or_equal(s(0), s(s(0))), less_or_equal(X, 0), ...
    ```
  
  ◦ Multiple solutions:
    ```prolog
    less_or_equal(X, s(0)), less_or_equal(s(s(0)), Y), etc.
    ```

- Addition:

  ```prolog
  plus(0, X, X) :- nat(X).
  plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
  ```

  ◦ Multiple uses (modes):
    ```prolog
    plus(s(s(0)), s(0), Z), plus(s(s(0)), Y, s(0))
    ```
  
  ◦ Multiple solutions:
    ```prolog
    plus(X, Y, s(s(s(0)))), etc.
    ```
Recursive Programming: Arithmetic

- Another possible definition of addition:

  ```prolog
  plus(X,0,X) :- nat(X).
  plus(X,s(Y),s(Z)) :- plus(X,Y,Z).
  ```

- The meaning of `plus` is the same, even if both definitions are combined.

- Not recommended: several proof trees for the same query → not efficient, not concise. We look for minimal axiomatizations.

- The art of logic programming: finding compact and computationally efficient formulations!

- Try to define: `times(X,Y,Z)` (\(Z = X \times Y\)), `exp(N,X,Y)` (\(Y = X^N\)), `factorial(N,F)` (\(F = N!\)), `minimum(N1,N2,Min)`, ...
Recursive Programming: Arithmetic

- Definition of \( \text{mod}(X, Y, Z) \)
  “Z is the remainder from dividing X by Y”

\[ \exists Q \text{s.t. } X = Y \ast Q + Z \land Z < Y \]

\[ \Rightarrow \]

\[ \text{mod}(X, Y, Z) :- \text{less}(Z, Y), \text{times}(Y, Q, W), \text{plus}(W, Z, X). \]

\[ \text{less}(\emptyset, s(X)) :- \text{nat}(X). \]

\[ \text{less}(s(X), s(Y)) :- \text{less}(X, Y). \]

- Another possible definition:

\[ \text{mod}(X, Y, X) :- \text{less}(X, Y). \]

\[ \text{mod}(X, Y, Z) :- \text{plus}(X_1, Y, X), \text{mod}(X_1, Y, Z). \]

- The second is much more efficient than the first one
  (compare the size of the proof trees).
Recursive Programming: Arithmetic/Functions

- The Ackermann function:
  \[
  \begin{align*}
  \text{ackermann}(0,N) & = N+1 \\
  \text{ackermann}(M,0) & = \text{ackermann}(M-1,1) \\
  \text{ackermann}(M,N) & = \text{ackermann}(M-1,\text{ackermann}(M,N-1))
  \end{align*}
  \]

- In Peano arithmetic:
  \[
  \begin{align*}
  \text{ackermann}(0,N) & = s(N) \\
  \text{ackermann}(s(M1),0) & = \text{ackermann}(M1,s(0)) \\
  \text{ackermann}(s(M1),s(N1)) & = \text{ackermann}(M1,\text{ackermann}(s(M1),N1))
  \end{align*}
  \]

- Can be defined as:
  \[
  \begin{align*}
  \text{ackermann}(0,N,s(N)). \\
  \text{ackermann}(s(M1),0,Val) & :- \text{ackermann}(M1,s(0),Val). \\
  \text{ackermann}(s(M1),s(N1),Val) & :- \text{ackermann}(s(M1),N1,Val1), \\
  & \quad \text{ackermann}(M1,Val1,Val).
  \end{align*}
  \]

- I.e., in general, \textit{functions} can be coded as a predicate with one more argument, which represents the output (and additional syntactic sugar often available).
Functional Syntax: Packages and Directives (I)

• :- use_package(fsyntax). Provides:

  • ~ “eval”, which makes the last argument implicit. This allows writing, e.g.
    \[
    \text{p}(X,Y) \leftarrow q(X,Z), r(Z,Y).
    \]
    as
    \[
    \text{p}(X,Y) \leftarrow r(\neg q(X),Y).
    \]
    or
    \[
    \text{p}(X,\neg r(\neg q(X))).
    \]

  • := for definitions: which allows writing, e.g.
    \[
    \text{p}(X,Y) \leftarrow q(X,Z), r(Z,Y).
    \]
    as
    \[
    \text{p}(X) := Y \leftarrow r(\neg q(X),Y).
    \]
    or
    \[
    \text{p}(X) := \neg r(\neg q(X)).
    \]

  • | for or, etc.
• Thus, we can now write:

\[ \text{ackmann}(s\ M, s\ N) := \neg\text{ackmann}(M, \neg\text{ackmann}(s\ M, N)) \].

• To evaluate automatically functors that are defined as functions (so there is no need to use \( \neg \) for them):

\[
\text{:- fun_eval ackmann/2.}
\]

\[ \text{ackmann}(s\ M, s\ N) := \text{ackmann}(M, \text{ackmann}(s\ M, N)) \].

• To enable this for all functions defined in a given file:

\[
\text{:- fun_eval defined(true).}
\]

• To evaluate arithmetic functors automatically (no need for \( \neg \) for them):

\[
\text{:- fun_eval arith(true).}
\]

\[ \text{add_one}(X,X+1) \].

• The functional package includes fsyntax + both fun_eval’s above:

\[
\text{:- use_package(functional).}
\]
The Ackermann function (Peano) in Ciao’s functional Syntax and defining \( s \) as a prefix operator:

\[
\begin{align*}
\text{:- use_package(functional).} \\
\text{:- op(500,fy,s).} \\
\text{ackermann( 0, N) := s N.} \\
\text{ackermann(s M, 0) := ackermann(M, s 0).} \\
\text{ackermann(s M, s N) := ackermann(M, ackermann(s M, N) ).}
\end{align*}
\]

Convenient in other cases – e.g. for defining types:

\[
\begin{align*}
nat(0). \\
nat(s(X)) :- nat(X).
\end{align*}
\]

Using special := notation for the “return” (last) the argument:

\[
\begin{align*}
nat := 0. \\
nat := s(X) := nat(X).
\end{align*}
\]
Moving body call to head using the ~ notation ("evaluate and replace with result"):

\[
\text{nat} := 0.
\text{nat} := s(\text{~nat}).
\]

"~" not needed with funционаl package if inside its own definition:

\[
\text{nat} := 0.
\text{nat} := s(\text{nat}).
\]

Using an \texttt{:- op} declaration to define \texttt{s} as a \textit{prefix operator}:

\[
\text{nat} := 0.
\text{nat} := s \text{ nat}.
\]

Using "|" (disjunction):

\[
\text{nat} := 0 \mid s \text{ nat}.
\]

Which is exactly equivalent to:

\[
\text{nat}(0).
\text{nat}(s(X)) :- \text{nat}(X).
\]
Recursive Programming: Lists

- Binary structure: first argument is *element*, second argument is *rest* of the list.

- We need:
  - A constant symbol: we use the *constant* \([\ ]\) (\(\rightarrow\) denotes the empty list).
  - A functor of arity 2: traditionally the dot 
    “.” (which is overloaded).

- Syntactic sugar: the term \(\cdot(X,Y)\) is denoted by \([X|Y]\) (\(X\) is the *head*, \(Y\) is the *tail*).

<table>
<thead>
<tr>
<th>Formal object</th>
<th>“Cons pair” syntax</th>
<th>“Element” syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(.a,[])</td>
<td>([a</td>
<td>[]])</td>
</tr>
<tr>
<td>(.a,.(b,[]))</td>
<td>([a</td>
<td>[b</td>
</tr>
<tr>
<td>(.a,.(b,(c,[]))))</td>
<td>([a</td>
<td>[b</td>
</tr>
<tr>
<td>(.a,X)</td>
<td>([a</td>
<td>X])</td>
</tr>
<tr>
<td>(.a,.(b,X))</td>
<td>([a</td>
<td>[b</td>
</tr>
</tbody>
</table>

- Note that:
  - \([a,b]\) and \([a|X]\) unify with \{\(X = [b]\)\}
  - \([a]\) and \([a|X]\) unify with \{\(X = []\)\}
  - \([a]\) and \([a,b|X]\) do not unify
  - \([]\) and \([X]\) do not unify
Recursive Programming: Lists (Contd.)

- Type definition (no syntactic sugar):
  
  ```
  list([]).
  list.(X,Y)) :- list(Y).
  ```

- Type definition, with some syntactic sugar ([ ] notation):
  
  ```
  list([]).
  list([X|Y]) :- list(Y).
  ```

- Type definition, using also functional package:
  
  ```
  list := [] | [_|list].
  ```

- “Exploring” the type:
  
  ```
  ?- list(L).
  L = []  ? ;
  L = [_]  ? ;
  L = [_,_]  ? ;
  L = [_,_,_]  ?
  ...
  ```
Recursive Programming: Lists (Contd.)

• X is a member of the list Y:
  
  member(a,[a]).  member(b,[b]).  etc.  ⇒  member(X,[X]).
  member(a,[a,c]).  member(b,[b,d]).  etc.  ⇒  member(X,[X,Y]).
  member(a,[a,c,d]).  member(b,[b,d,l]).  etc.  ⇒  member(X,[X,Y,Z]).

  ⇒  member(X,[X|Y]) :- list(Y).

  member(a,[c,a]),  member(b,[d,b]).  etc.  ⇒  member(X,[Y,X]).
  member(a,[c,d,a]).  member(b,[s,t,b]).  etc.  ⇒  member(X,[Y,Z,X]).

  ⇒  member(X,[Y|Z]) :- member(X,Z).

• Resulting definition:

  member(X,[X|Y]) :- list(Y).
  member(X,[_|T]) :- member(X,T).

• Uses of member(X,Y):
  ◦ checking whether an element is in a list (member(b,[a,b,c]))
  ◦ finding an element in a list (member(X,[a,b,c]))
  ◦ finding a list containing an element (member(a,Y))
Recursive Programming: Lists (Contd.)

- Combining lists and naturals:
  - Computing the length of a list:
    ```prolog
    len([], 0).
    len([H|T], s(LT)) :- len(T, LT).
    ```
  - Adding all elements of a list:
    ```prolog
    sumlist([], 0).
    sumlist([H|T], S) :- sumlist(T, ST), plus(ST, H, S).
    ```
  - The type of lists of natural numbers:
    ```prolog
    natlist([]).
    natlist([H|T]) :- nat(H), natlist(T).
    or:
    natlist := [] | [^nat|natlist].
    ```
Exercises:

- Define: \texttt{prefix}(X, Y) (the list \(X\) is a prefix of the list \(Y\)), e.g.
  \texttt{prefix}([a, b], [a, b, c, d])
- Define: \texttt{suffix}(X, Y), \texttt{sublist}(X, Y), ...
• Concatenation of lists:
  ◇ Base case:
  \[
  \text{append}([], [a], [a]). \quad \text{append}([], [a,b], [a,b]). \quad \text{etc.}
  \]
  \[
  \Rightarrow \text{append}([], Ys, Ys) :- \text{list}(Ys).
  \]
  ◇ Rest of cases (first step):
  \[
  \text{append}([a], [b], [a,b]).
  \text{append}([a], [b,c], [a,b,c]). \quad \text{etc.}
  \]
  \[
  \Rightarrow \text{append}([X], Ys, [X|Ys]) :- \text{list}(Ys).
  \]
  \[
  \text{append}([a,b], [c], [a,b,c]).
  \text{append}([a,b], [c,d], [a,b,c,d]). \quad \text{etc.}
  \]
  \[
  \Rightarrow \text{append}([X,Z], Ys, [X,Z|Ys]) :- \text{list}(Ys).
  \]
  This is still infinite \(\Rightarrow\) we need to generalize more.
Recursive Programming: Lists (Contd.)

- Second generalization:

  \[
  \text{append}([X], Ys, [X|Ys]) :\text{-} \text{list}(Ys).
  \]
  \[
  \text{append}([X,Z], Ys, [X,Z|Ys]) :\text{-} \text{list}(Ys).
  \]
  \[
  \text{append}([X,Z,W], Ys, [X,Z,W|Ys]) :\text{-} \text{list}(Ys).
  \]

  \[
  \Rightarrow \text{append}([X|Xs], Ys, [X|Zs]) :\text{-} \text{append}(Xs, Ys, Zs).
  \]

- So, we have:

  \[
  \begin{align*}
  \text{append}([], Ys, Ys) :\text{-} & \text{list}(Ys). \\
  \text{append}([X|Xs], Ys, [X|Zs]) :\text{-} & \text{append}(Xs, Ys, Zs).
  \end{align*}
  \]

- Another way of reasoning: thinking inductively.

  - The base case is:  \text{append}([], Ys, Ys) :\text{-} \text{list}(Ys).
  
  - If we assume that \text{append}(Xs, Ys, Zs) works for some iteration, then, in the next one, the following holds: \text{append}([X|Xs], Ys, [X|Zs]).
Uses of append:

- Concatenate two given lists:
  
  \[
  \text{?- } \text{append}([a,b,c],[d,e],L).
  \]
  
  \[
  L = [a,b,c,d,e] \ ?
  \]

- Find differences between lists:
  
  \[
  \text{?- } \text{append}(D,[d,e],[a,b,c,d,e]).
  \]
  
  \[
  D = [a,b,c] \ ?
  \]

- Split a list:
  
  \[
  \text{?- } \text{append}(A,B,[a,b,c,d,e]).
  \]
  
  \[
  A = [],
  \]
  
  \[
  B = [a,b,c,d,e] \ ? ;
  \]
  
  \[
  A = [a],
  \]
  
  \[
  B = [b,c,d,e] \ ? ;
  \]
  
  \[
  A = [a,b],
  \]
  
  \[
  B = [c,d,e] \ ? ;
  \]
  
  \[
  A = [a,b,c],
  \]
  
  \[
  B = [d,e] \ ?
  \]
  
  \[
  \ldots
  \]
Recursive Programming: Lists (Contd.)

- **reverse(Xs,Ys):** Ys is the list obtained by reversing the elements in the list Xs. Each element X of \([X|Xs]\) should end up at the end of the reversed version of Xs:
  
  \[
  \text{reverse}([X|Xs], Ys) : - \\
  \text{reverse}(Xs, Zs), \\
  \text{append}(Zs, [X], Ys).
  \]

  Inductively: if we assume Xs is already reversed as Zs, if Xs has one more element at the beginning, it goes at the end of Zs.

  How can we stop (i.e., what is our base case):

  \[
  \text{reverse}([], []). 
  \]

- As defined, reverse(Xs,Ys) is very inefficient. Another possible definition:
  (uses an *accumulating parameter*)

  \[
  \text{reverse}(Xs, Ys) : - \text{reverse}(Xs, [], Ys). \\
  \text{reverse}([], Ys, Ys). \\
  \text{reverse}([X|Xs], \text{Acc}, Ys) : - \text{reverse}(Xs, [X|\text{Acc}], Ys). 
  \]

  \Rightarrow \text{Find the differences in terms of efficiency between the two definitions.}
Recursive Programming: Binary Trees

- Represented by a ternary functor \( \text{tree}(\text{Element}, \text{Left}, \text{Right}) \).
- Empty tree represented by \text{void}.
- Definition:

```prolog
binary_tree(void).
binary_tree(tree(_Element, \text{Left}, \text{Right})) :-
    binary_tree(Left),
    binary_tree(Right).
```

- Defining \text{tree_member}(\text{Element}, \text{Tree}):  

```prolog
tree_member(X, tree(X, \text{Left}, \text{Right})) :-
    binary_tree(Left),
    binary_tree(Right).
tree_member(X, tree(_, \text{Left}, \text{Right})) :- tree_member(X, Left).
tree_member(X, tree(_, \text{Left}, \text{Right})) :- tree_member(X, Right).
```
• Defining `pre_order(Tree,Elements)`: Elements is a list containing the elements of Tree traversed in *preorder*.

```
pre_order(void,[],).
predorder(tree(X,Left,Right),Elements) :-
    pre_order(Left,ElementsLeft),
    pre_order(Right,ElementsRight),
    append([X|ElementsLeft],ElementsRight,Elements).
```

• Exercise – define:

  ◦ `in_order(Tree,Elements)`
  ◦ `post_order(Tree,Elements)`
Polymorphism

- Note that the two definitions of `member/2` can be used simultaneously:

```
lt_member(X,[X|Y]) :- list(Y).
lt_member(X,[_|T]) :- lt_member(X,T).
```

```
lt_member(X,tree(X,L,R)) :- binary_tree(L), binary_tree(R).
lt_member(X,tree(Y,L,R)) :- lt_member(X,L).
lt_member(X,tree(Y,L,R)) :- lt_member(X,R).
```

Lists only unify with the first two clauses, trees with clauses 3–5!

- `:- lt_member(X,[b,a,c]).`
  
  `X = b ; X = a ; X = c`

- `:- lt_member(X,tree(b,tree(a,void,void),tree(c,void,void))).`
  
  `X = b ; X = a ; X = c`

- Also, try (somewhat surprising): `:- lt_member(M,T).`
Recognizing (and generating!) polynomials in some term X:

- X is a polynomial in X
- a constant is a polynomial in X
- sums, differences and products of polynomials in X are polynomials
- also polynomials raised to the power of a natural number and the quotient of a polynomial by a constant

```prolog
polynomial(X,X).
polynomial(Term,X) :- pconstant(Term).
polynomial(Term1+Term2,X) :- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1-Term2,X) :- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1*Term2,X) :- polynomial(Term1,X), polynomial(Term2,X).
polynomial(Term1/Term2,X) :- polynomial(Term1,X), pconstant(Term2).
polynomial(Term1^N,X) :- polynomial(Term1,X), nat(N).
```
Recursive Programming: Manipulating Symb. Expressions (Contd.)

- **Symbolic differentiation: deriv(Expression, X, Derivative)

```
deriv(X,X,s(0)).
deriv(C,X,0) :- pconstant(C).
deriv(U+V,X,DU+DV) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U-V,X,DU-DV) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U*V,X,DU*V+U*DV) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U/V,X,(DU*V-U*DV)/V^s(s(0))) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U^s(N),X,s(N)*U^N*DU) :- deriv(U,X,DU), nat(N).
deriv(log(U),X,DU/U) :- deriv(U,X,DU).
...```

- `?- deriv(s(s(s(0)))*x+s(s(0)),x,Y).`

- A simplification step can be added.
Recursive Programming: Graphs

- A common approach: make use of another data structure, e.g., lists:
  - Graphs as lists of edges.
- Alternative: make use of Prolog's program database:
  - Declare the graph using facts in the program.
    ```prolog
    edge(a,b).  edge(c,a).
    edge(b,c).  edge(d,a).
    ```
- Paths in a graph: `path(X,Y)` iff there is a path in the graph from node \( X \) to node \( Y \).
  ```prolog
  path(A,B) :- edge(A,B).
  path(A,B) :- edge(A,X), path(X,B).
  ```
- Circuit: a closed path. `circuit` iff there is a path in the graph from a node to itself.
  ```prolog
  circuit :- path(A,A).
  ```
Recursive Programming: Graphs (Exercises)

• Modify `circuit/0` so that it provides the circuit. (You have to modify also `path/2`)
• Propose a solution for handling several graphs in our representation.
• Propose a suitable representation of graphs as data structures.
• Define the previous predicates for your representation.

• Consider unconnected graphs (there is a subset of nodes not connected in any way to the rest) versus connected graphs.
• Consider directed versus undirected graphs.

• Try `path(a,d)`. Solve the problem.
• Recognizing the sequence of characters accepted by the following *non-deterministic, finite automaton* (NDFA):  

```
q0
a
q1
b
```

where \( q0 \) is both the *initial* and the *final* state.

• Strings are represented as lists of constants (e.g., \([a,b,b]\)).

• Program:

```prolog
initial(q0).  delta(q0,a,q1).
delta(q0,a,q1).
delta(q1,b,q0).
final(q0).  delta(q1,b,q1).
accept(S) :- initial(Q), accept_from(S,Q).
accept_from([],Q) :- final(Q).
accept_from([X|Xs],Q) :- delta(Q,X,NewQ), accept_from(Xs,NewQ).
```
A nondeterministic, stack, finite automaton (NDSFA):

\[
\text{accept}(S) \leftarrow \text{initial}(Q), \text{accept}_{\text{from}}(S,Q,[]).
\]

\[
\text{accept}_{\text{from}}([],Q,[]) \leftarrow \text{final}(Q).
\]

\[
\text{accept}_{\text{from}}([X|Xs],Q,S) \leftarrow \text{delta}(Q,X,S,\text{NewQ},\text{NewS}),
\quad \text{accept}_{\text{from}}(Xs,\text{NewQ},\text{NewS}).
\]

\[
\text{initial}(q0).
\]

\[
\text{final}(q1).
\]

\[
\text{delta}(q0,X,Xs,q0,[X|Xs]).
\]

\[
\text{delta}(q0,X,Xs,q1,[X|Xs]).
\]

\[
\text{delta}(q0,X,Xs,q1,Xs).
\]

\[
\text{delta}(q1,X,[X|Xs],q1,Xs).
\]

What sequence does it recognize?
Recursive Programming: Towers of Hanoi

- **Objective:**
  - Move tower of N disks from peg a to peg b, with the help of peg c.

- **Rules:**
  - Only one disk can be moved at a time.
  - A larger disk can never be placed on top of a smaller disk.

\[ N = 1 \]
\[
\begin{array}{c}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

\[ N = 2 \]
\[
\begin{array}{c}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

\[ N = 3 \]
\[
\begin{array}{c}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]
We will call the main predicate `hanoi_moves(N,Moves)`

- $N$ is the number of disks and $Moves$ the corresponding list of “moves”.
- Each move `move(A, B)` represents that the top disk in $A$ should be moved to $B$.

*Example*: The moves for three disks

\[
\text{hanoi_moves( } s(s(s(0))),
\begin{array}{l}
\quad [\text{move(a,b), move(a,c), move(b,c), move(a,b), } \\
\quad \text{move(c,a), move(c,b), move(a,b)} ]\end{array}
\]
Recursive Programming: Towers of Hanoi (Contd.)

- A general rule: To move $N$ disks from peg A to peg B using peg C we need to:

  move $N-1$ disks to peg C using peg B, move the bottom disk to peg B, and then move the $N-1$ disks from peg C to peg B using peg A.

- We capture this in a predicate $\text{hanoi}(N, \text{Orig}, \text{Dest}, \text{Help}, \text{Moves})$ where

  "Moves contains the moves needed to move a tower of $N$ disks from peg Orig to peg Dest, with the help of peg Help."

  \[
  \begin{align*}
  &\text{hanoi}(s(0), \text{Orig}, \text{Dest}, \_\text{Help}, [\text{move}(\text{Orig}, \text{Dest})]). \\
  &\text{hanoi}(s(N), \text{Orig}, \text{Dest}, \text{Help}, \text{Moves}) :- \\
  &\quad \text{hanoi}(N, \text{Orig}, \text{Help}, \text{Dest}, \text{Moves1}), \\
  &\quad \text{hanoi}(N, \text{Help}, \text{Dest}, \text{Orig}, \text{Moves2}), \\
  &\quad \text{append}(\text{Moves1}, [\text{move}(\text{Orig}, \text{Dest}) | \text{Moves2}], \text{Moves}).
  \end{align*}
  \]

- And we simply call this predicate:

  \[
  \begin{align*}
  &\text{hanoi\_moves}(N, \text{Moves}) :- \\
  &\quad \text{hanoi}(N, \text{a}, \text{b}, \text{c}, \text{Moves}).
  \end{align*}
  \]
Learning to Compose Recursive Programs

- To some extent it is a simple question of practice.
- By generalization (as in the previous examples): elegant, but sometimes difficult? (Not the way most people do it.)
- Think inductively: state first the base case(s), and then think about the general recursive case(s).
- Sometimes it may help to compose programs with a given use in mind (e.g., “forwards execution”), making sure it is declaratively correct. Consider then also if alternative uses make sense.
- Sometimes it helps to look at well-written examples and use the same “schemas.”
- Using a global top-down design approach can help (in general, not just for recursive programs):
  ◦ State the general problem.
  ◦ Break it down into subproblems.
  ◦ Solve the pieces.
- Again, the best approach: practice, practice, practice.